# The Power of Rich Syntax for Model-based Development

Ethan K. Jackson, Wolfram Schulte
Microsoft Research,
One Microsoft Way, Redmond, WA
{ejackson,schulte}@microsoft.com

Janos Sztipanovits
Institute for Software Integrated Systems,
Vanderbilt University, Nashville, TN
janos.sztipanovits@vanderbilt.edu

## Abstract

*During the last century, many general purpose programming languages have been developed, all having rigid syntax and often a von-Neuman view of the world. With the rise of model-based development this changes: Feature-oriented programming, domain specific languages, and platform-based design use rich and custom syntaxes to capture domain specific abstractions, refinement mappings, and design spaces. In this paper we show how a formalization of rich syntax can be used to compose abstractions, validate refinement maps, and construct design spaces. We describe a tool* FORMULA *for computing these properties, and present a series of examples from automotive embedded systems.*

## 1. Introduction

Research in model-based development has produced new approaches to architecting and modularizing software systems. Promising approaches include: model-driven architecture [45], platform-based design [1], domain-specific languages [22], feature-oriented design [51] and aspect-oriented programming [31]. Though many of these approaches overlap [34], each offers unique strengths from a unique perspective. However, all of these perspectives makes it unclear how to compare, reuse, and generalize accomplishments in model-based development.

We observe that a key commonality exists across the spectrum of approaches: Design artifacts are captured using (1) rich and customizable syntactic constructs and (2) expressive constraints over syntax. By *rich syntactic constructs* we mean notations that are (at least) set-based and relational, e.g. graph-theoretic [19]. By *expressive constraints* we mean well-formedness rules limiting syntax via type-constraints and context-dependent invariants, e.g. *OCL* [46]. We use the term *rich syntax* as a loose shorthand for these properties. Unlike traditional programming languages where syntax is fixed for all problems, model-based approaches encourage rich custom syntaxes to be created on a *per problem* basis. This new dimension of expressiveness allows engineers to capture design invariants and abstractions using custom expressive notations.

In this paper we provide a novel framework for rich syntax arising in model-based development. We define:

- *domains* as a formalization of rich syntax, utilizing extended Horn logic to represent the constructs and invariants of modeling artifacts (Section 3).
- *composition operators* for building new domains from existing ones. We also check that compositions do not contain contradictions (Section 4).
- *transformations* as translation procedures between domains. We check that transformations are well-behaved, i.e. well-formed inputs are always translated to well-formed outputs (Section 5).
- *design spaces* as sets of syntactic instances over domains. Design spaces can be compactly represented and elements can be enumerated (Section 6).

We have developed a tool called FORMULA that implements these tasks. This implementation uses a *model generation* procedure to construct sets of rich syntax satisfying key properties [26]. The examples in this paper are written in the notation of FORMULA.

## 2. Rich Syntax in Model-based Development

The automotive community has been an important earlier adopter of modeling technology[52, 60]. Many modeling approaches converge in the automotive domain, making it an ideal place to study modeling foundations. In this section we introduce three automotive examples, each of which is constructed using a different modeling technique. Along the way, we show that *rich syntax* spans all of these modeling styles, providing a common substrate for composition and analysis. This overview focuses on rich syntax, and does not attempt to be exhaustive in any sense.

### 2.1 Features and Aspects

*Feature-oriented design* [51, 6] and *aspect-oriented programming* [31] are attempts to modularize and reuse software components even when the components have complex coupling between them. Feature-oriented design partitions a system into *features*, where each feature represents some slice of the overall system. Distinct features utilize overlapping parts of the implementation (e.g. classes, components, etc...) and thus have coupling between them. The goal of feature-oriented design is to mix and match features to create variants (product lines) of a large software system. This vision requires mechanisms to reason about which sets of features are compatible.

A *feature diagram* abstracts the coupling between features allowing the engineer to reason about the possible system variants. Figure 1 shows some features in a car. The diagram describes a tree of features and the interactions between features. For example, the car feature must have a maneuvering feature, but cruise control is optional. The acceleration feature requires exactly one of the manual or automatic features. If cruise control is in the car then the automatic feature must be selected.

Let $F$ be a set of features, then a legal program variant $V \subseteq F$ is a subset of $F$ satisfying the feature diagram. It was observed in [7] that each feature diagram induces a BNF grammar, and this provides a formal basis for checking if a program variant $V$ is legal. Table 1 shows a partial grammar induced by the car feature diagram. Leaf nodes in the diagram become terminal tokens in the grammar (written in upper-case); internal nodes are non-terminals (written in lower-case).

More complex forms of coupling are difficult to capture if feature diagrams are formalized as BNF grammars. For example, the implication from Cruise Control to Automatic transmission is problematic. Consequently, various extensions and formalizations have been pursued [14]. For instance, in [36] feature diagrams are reduced to first-order propositional formulas permitting Boolean constraints between features. This is an exemplar of rich syntax in model-based development.

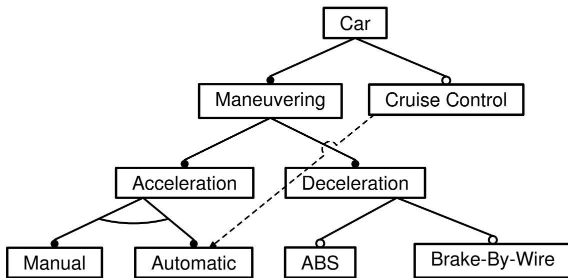



**Figure 1. Feature diagram of a car's subsystems.**

| | |
|---|---|
| car: | maneuvering [CRUISE CONTROL] |
| maneuvering: | acceleration deceleration |
| acceleration: | MANUAL \| AUTOMATIC |
| deceleration: | [ABS] [BREAK-BY-WIRE] |

**Table 1. Feature diagram as a grammar.**

### 2.2 Domain-specific Languages

*Domain-specific languages* (DSLs) evolved from the observation that software development is application-specific and the application context imposes strong constraints on the structure of software [28, 54]. DSLs capture domain-specific concepts, and come in two parts: The *abstract syntax*, which is a rich syntax, and the *behavioral semantics*, which formally assigns behaviors to instances of the abstract syntax. The behavioral semantics is often expressed using a mathematical description of behaviors and can also be domain-specific [12].

For example, in the automotive domain software executes on embedded processors called *ECUs* (electronic control units). ECUs communicate with each other through local-area networks, called *buses*, that have strict rules on the structure, duration, and frequency of *messages* sent between ECUs. Any software system must take such application-specific constraints into account. Figure 2 shows a *metamodel* describing the abstract syntax for an automotive DSL. The diagram is an extended type of *UML diagram*, called a MOF (meta-object facility) metamodel [44]. It explains that a Car is a basic concept parameterized by data indicating the make and model. Every Car contains one or more ECU entities, and each ECU may contain Message entities. A Car contains Buses that link ECUs together. Buses come in two varieties: CAN and FlexRay [55]. Additionally, metamodels can be annotated with *constraints*, often written in the *object constraint language* (OCL). For example, we may add the OCL-like constraint:

$$\forall\, \mathsf{ECU}\; e,\ \forall\, \mathsf{Bus}\; b,\ (e.isCritical \wedge b.dst = e) \Rightarrow b\, \mathsf{is\, CAN}$$

i.e. every ECU containing a safety-critical task communicates on a CAN bus. Loosely speaking, an instance of the abstract syntax is a set of entities that conform to the metamodel, including its constraints.

This simple example shows that rich syntaxes are prevalent in DSLs. At first glance, DSLs resemble graph-theoretic objects. For example, ECUs are vertex-like and Buses are edge-like. This has lead to significant work in extending graph-grammars to capture the rich syntax of DSLs

[8, 48]. These extensions must address data-types, containment hierarchies, non-binary relations, and hierarchy-crossing constructs.

## 2.3 Platform-based Design and Model-Driven Architecture

*Platform-based design* and *Model-Driven Architecture* (MDA) focus on migrating abstract specifications to implementations through incremental translation steps [45, 1]. The engineer begins by developing a *functional model* of the system, also called a *platform-independent model* (PIM) in MDA. The functional model is an abstract description of *what the system should do*. Along side the functional model is an *architectural model* specifying *what the system can do*. The goal of platform-based design is to find an appropriate *platform mapping* (translation) from the function to the architecture so that the system is correctly implemented [50].

The left-hand side of Figure 3 shows a functional model, architectural model, and platform mapping. The functional model consists of three tasks (gray circles) S,T, and U. The undirected edges between the tasks are resource constraints. A pair of tasks with resource constraints cannot execute on the same processor, because local resources (e.g. memory capacity) are insufficient to support both tasks. The architectural model consists of three processors (squares) P1, P2, and P3 indicating the processing capability of the implementation. The platform mapping must place tasks onto processors without violating resource constraints. The figure shows one such mapping. In this example finding a platform mapping is equivalent to the NP-hard coloring problem. This is typical in platform-based design, and is a manifestation of the difficulty in changing abstraction levels. (This example is adapted from the problem of scheduling tasks with *conflict graphs* over multiple processors [38].)

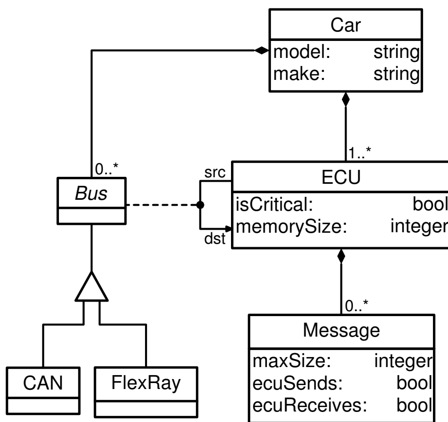



**Figure 2. Metamodel of ECU/bus architecture.**

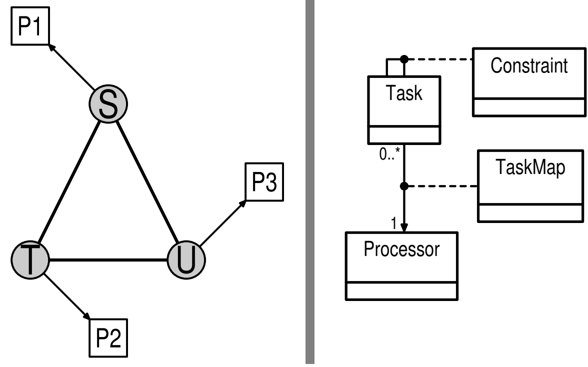



**Figure 3. (Left) Function, architecture, and platform mapping, (Right) Metamodel of problem domain.**

Rich syntax also plays a key role in platform-based design. It is used to describe the functional and architectural abstraction layers, and to define the valid platform mappings. The right-hand side of Figure 3 shows the metamodel defining exactly this problem domain. According to this metamodel, Tasks are connected by resource Constraints and each Task is mapped to exactly one Processor. This metamodel also has a constraint (not shown in figure) that two tasks connected by Constraint edges cannot be mapped to the same Processor. This well-formedness rule is expressed within the rich syntax, and does not require any knowledge of the computations performed by tasks and processors. The set of all instances conforming to this metamodel is exactly the set of possible functions/architectures with valid mappings.

The observation that the legal syntactic instances correspond to behaviorally meaningful designs has lead to new techniques for *design space exploration* [9]. For example, given a set of tasks $T$ with resource constraints, then there is an associated set of architectures that admit valid platform mappings. This design space of legal architectures/mappings can be characterized once the rich syntax is formalized. Design-space exploration using rich syntax has been realized for specific problem domains [43, 29].

## 2.4 Rich Syntaxes for Composition

We have shown that rich syntaxes span model-based development. More importantly, rich syntax can be used to compose modeling approaches. Figure 4 illustrates how development styles realistically interact through rich syntax. This figure shows an instance of Car from the DSL presented in Figure 2. The squares labeled $E_{1,2,...}$ are instances of ECUs connected by Buses. (Messages and other data are not shown.) The network of ECUs and Buses forms

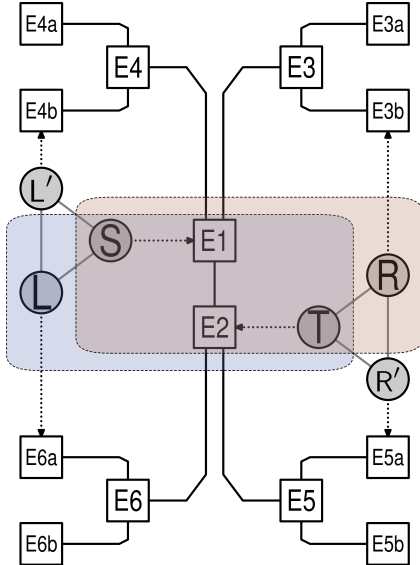



**Figure 4. Composition of modeling styles through rich syntax.**

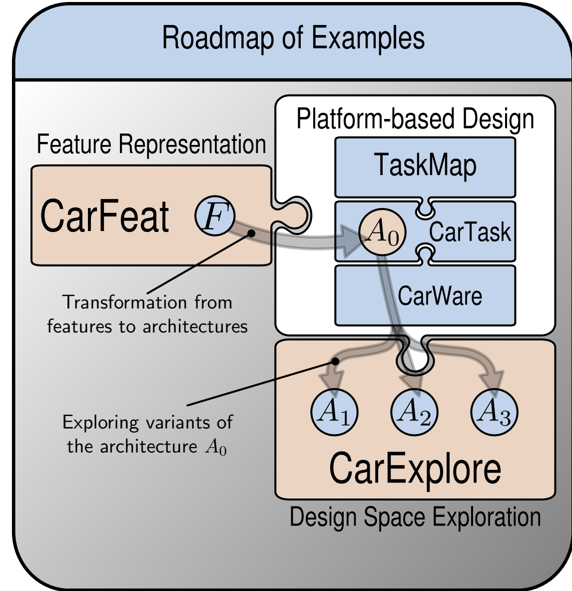



**Figure 5. Overview of examples**

an architectural model, in the sense of platform-based design. Any functionally correct software system must respect this architecture. The arrows show the platform mapping of Tasks to Processors, where the Processors are actually ECUs.

The shaded areas show subfeatures of the Cruise Control feature. The tasks L, R correspond to sensors on the left/right side of the vehicle. The task S calculates engine dynamics, while T is the main controller used by the cruise control. The two subfeatures span the implementation and overlap at their intersection: $\{S, T, E_1, E_2\}$.

Though composition of modeling styles seems natural, there are theoretical issues that must be resolved. First, consider the "replacement" of Processors with ECUs. This replacement binds the two rich syntaxes in a non-trivial way: ECUs inherit constraints on Processors and vice-versa. In general, this composition may be ill-defined, i.e. no legal instance exists in the composition. Second, syntaxes can be related in more complex ways than replacement, i.e. by transforming from one syntax to another. However, transformations may themselves contain mistakes in the form of nonsensical rewrites. For example, we must ensure that any transformation process always produces well-scheduled Tasks. Third, modeling provides a means to explore architectures before implementation. For instance, the possible schedules of Tasks to ECUs represents a large space of architectural variants. The challenge is to compactly represent and explore these design choices.

### 2.5 Roadmap of Examples

In the remainder of this paper we develop the three preceding examples, initially given in isolation, using a unified framework enabling early end-to-end analysis. These examples are expressed in the language of FORMULA (**For**mal **M**odeling **U**sing **L**ogic **A**nalysis). FORMULA is a new tool using Horn logic extended with stratified negation as a basis for integrating and analyzing rich syntaxes and transformations occurring in model-based development [26]. Figure 5 shows the parts that we develop and their interrelationships. We begin by developing a rich syntax for *platform-based design of automotive systems*, as shown in the figure by the block labeled Platform-based Design. First, we define a generic platform devoid of any automotive-specific concepts. This is formalized with the TaskMap domain. Next, we construct a domain, called CarWare, for automotive-specific hardware architectures that does not support platform-based design. Third, these two domains are composed to form the CarTask domain, which can be used to perform automotive-specific platform-based design. This example shows how composition and theorem proving can be used to incrementally build domains with known properties.

The second example builds a feature language for automotive systems. (See the CarFeat block in Figure 5.) The CarFeat domain captures the grammar of the feature diagram in Figure 1. Next, a transformation (syntax translator) called FeatMap is defined, which converts feature sets into partial architectures. In the figure a feature set $F$ it translated into a partial architecture $A_0$. This example shows

how to construct transformations and prove that certain errors are absent from the transformations.

The third example uses FORMULA for design space exploration. We define the CarExplore domain as a subset of the legal CarTask instances. CarExplore instances correspond to a set of interesting architectural variants for automotive embedded systems. All of these examples are linked together to perform design space exploration for a particular feature set. In the figure $A_1$, $A_2$, and $A_3$ are architectural variants of $A_0$ that were calculated by FORMULA. This realizes the vision of Figure 4, which integrates many different modeling approaches in a consistent manner.

# 3 Domains and Model Generation

## 3.1 Elements of Rich Syntax

We have shown that rich syntaxes are built from datatypes, relations over sets/relations, and expressive context-sensitive constraints. We formalize rich syntax by:

1. Using *terms* (i.e. uninterpreted function symbols and typed constants) to encode problem-specific sets and relations,
2. Capturing language invariants with an extended form of *Horn* logic,
3. Choosing the terms and Horn axioms on a per syntax basis.

We illustrate this approach by formalizing the rich syntax of the task mapping language (Figure 3).

### 3.1.1 Signatures and Terms

A function symbol, e.g. $f(\cdot)$, is a symbol denoting a unary function over a universe $U$. An *uninterpreted function symbol* is a symbol standing for a function that satisfies no additional equalities other than $\forall x \in U,\ f(x) = f(x)$. Let $\Sigma$ be an infinite alphabet of constants, then a *term* is either a constant or an application of some uninterpreted function symbol to a term. For example, $\{\mathtt{1}, f(\mathtt{2}), f(f(\mathtt{3}))\}$ are all terms assuming $\mathbb{Z} \subset \Sigma$. (The `typewriter` font denotes elements of $\Sigma$.) Henceforth, our function symbols will be $n$-ary to capture relations and other constructs. Constructing terms generalizes for arbitrary arity.

Uninterpreted function symbols form a flexible mechanism for capturing sets, relations, and relations over relations without assigning any deeper interpretations (semantics) to the syntax. A signature $\Upsilon$ is a finite set of $n$-ary function symbols. The *term algebra* $\mathcal{T}_\Upsilon(\Sigma)$ is an algebra where all symbols of $\Upsilon$ are uninterpreted, symbols can be applied to terms, and functions are one-to-one with disjoint images.

For example, the task mapping language can be encoded using the following signature:

$$\Upsilon_{task} = \left\{ \begin{array}{c} task(\cdot), processor(\cdot), \\ constraint(\cdot,\cdot), taskmap(\cdot,\cdot) \end{array} \right\} \quad (1)$$

The $constraint(\ldots)$ symbol is used to encode a binary relation over tasks, while the $taskmap(\ldots)$ symbol is a relation over tasks and processors. Sets of terms built from $\Upsilon_{task}$ encode syntactic instances of the task mapping language. For example, the set $M_{triangle}$ encodes the task scheduling instance of Figure 3:

$$\left\{ \begin{array}{c} task(\mathtt{S}), task(\mathtt{T}), task(\mathtt{U}), \\ processor(\mathtt{P}_1), processor(\mathtt{P}_2), processor(\mathtt{P}_3), \\ constraint(\mathtt{S},\mathtt{T}), constraint(\mathtt{S},\mathtt{U}), constraint(\mathtt{T},\mathtt{U}), \\ taskmap(\mathtt{S},\mathtt{P}_1), taskmap(\mathtt{T},\mathtt{P}_2), taskmap(\mathtt{U},\mathtt{P}_3) \end{array} \right\} \quad (2)$$

We have chosen one way of writing relations, but the term algebra mechanism supports flexible notations. For example, $constraint(task(\mathtt{S}), task(\mathtt{T}))$ builds constraint terms directly over task terms. This flexibility is used to easily build relations over relations. We call $\Upsilon$ the *primitive signature*, as its function symbols encode the syntactic primitives of some problem-specific syntax. The set of all syntactic instances of the rich syntax is the powerset of the term algebra: $\mathcal{P}(\mathcal{T}_\Upsilon(\Sigma))$.

### 3.1.2 Terms With Types

The powerset of terms contains many malformed instances of syntax. For example, the term

$$taskmap(constraint(\mathtt{1},\mathtt{2}), task(\mathtt{3}))$$

never belongs to a meaningful task mapping instance. We can refine the legal sets of terms by typing the arguments that can appear in function symbols. We call the structure

$$\Sigma_\subseteq = \langle I, \preceq, (\Sigma_i)_{i \in I} \rangle \quad (3)$$

an *order sorted* alphabet $\Sigma_\subseteq$. The set $I$, called the *index set*, is a set of sort names (alphabet names). Associated with each sort name $i \in I$ is a set of constants $\Sigma_i$ called the *carrier* of $i$. An order sorted alphabet has the following properties:

$$\Sigma = \bigcup_{i \in I} \Sigma_i, \quad (i \preceq j) \Leftrightarrow (\Sigma_i \subseteq \Sigma_j) \quad (4)$$

In other words, $\Sigma$ is the union of smaller alphabets and alphabets are ordered by set inclusion; the sub-typing relation $\preceq$ is set inclusion. A *type* $\tau$ is a term constructed from function symbols and elements of $I$ or the special top type $\top$. Each type $\tau$ identifies a subset $[\![\tau]\!] \subseteq \mathcal{T}_\Upsilon(\Sigma)$ according to:

```
TaskMap Specification - Part 1

1: domain TaskMap {
2:   primitives {
3:     task(ID), processor(ID),
4:     constraint(ID, ID),
5:     taskmap(ID, ID)
6:   } ...
```

**Figure 6. Specification of the TaskMap primitives with FORMULA**

1. The top type is the entire term algebra:

$$[\![\top]\!] = \mathcal{T}_{\Upsilon}(\Sigma) \tag{5}$$

2. A sort name $\tau \in I$ is just the carrier set $\Sigma_\tau$:

$$\forall \tau \in I,\ [\![\tau]\!] = \Sigma_\tau \tag{6}$$

3. Otherwise $\tau = f(\tau_1, \tau_2, \ldots, \tau_n)$ where $f$ is an $n$-ary function symbol:

$$[\![\tau]\!] = \left\{ t \in \mathcal{T}_{\Upsilon}(\Sigma) \,\middle|\, \begin{array}{c} t = f(t_1, t_2, \ldots, t_n) \wedge \\ \bigwedge_{1 \leq j \leq n} t_j \in [\![\tau_j]\!] \end{array} \right\} \tag{7}$$

The sub-typing relation $\preceq$ is extended to arbitrary types:

$$\forall \tau_p, \tau_q \quad (\tau_p \preceq \tau_q) \Leftrightarrow ([\![\tau_p]\!] \subseteq [\![\tau_q]\!]) \tag{8}$$

Figure 6 shows a partial specification of the TaskMap language in our FORMULA notation. Line 1 declares the TaskMap rich syntax; this declaration will be described in more detail later. Line 2 starts the list of primitive types. Line 3 simultaneously declares $task$ and $processor$ as unary function symbols with legal types $task(\texttt{ID})$ and $processor(\texttt{ID})$. The sort name `ID` contains elements used for unique identifiers. FORMULA is predefined with the sorts: `Real`, `Integer`, `PosInt`, `Natural`, `NegInt`, `String`, `ID`, and `Bool`. Lines 4 and 5 declare the $constraint$ and $taskmap$ symbols as binary relations over IDs. If $P$ is a set of primitive types, then the set $S(P)$ of possible syntactic instances is:

$$S(P) = \mathcal{P}\left( \bigcup_{\tau \in P} [\![\tau]\!] \right) \tag{9}$$

This refinement immediately eliminates many of the useless terms. Note, that all the functions are uninterpreted, so they are total over their types.

### 3.1.3 Expressive Constraints

Rich syntaxes often contain complex rules that cannot be captured by simple type-systems. One common solution to this problem is to provide a language for expressing syntactic rules. Unlike other approaches, we choose an extension of *Horn logic* to capture syntactic rules because:

1. Horn clauses are similar to pattern matches over sets of terms. It is easy to convert syntactic rules to Horn clauses.

2. Unlike other formalisms (e.g. OCL) our style of Horn axioms always terminate, so syntactic well-formedness is decidable.

3. Horn logic extended with stratified negation has expressive power [15].

In the interest of space we describe, but do not fully formalize Horn logic here. See [26] for a full set of definitions.

A *Horn clause* is a pair $(h, T)$. The *head* $h$ is a term with variables, and the *tail* (or body) $T = \{t_1, t_2, \ldots, t_n\}$ is a set of terms with variables. Variables are special constants from a set $V$ that can be replaced with other terms. (Assume $V \cap \Sigma = \emptyset$.) The semantics of Horn logic evaluates a set clauses $\Theta$ over a set of terms $X$ as follows: The variables in the tail of a clause are substituted with terms so that they occur in $X$. For each substitution, an occurrence of $h$ is added to $X$.

Consider the following Horn clause written in the standard notation $h \leftarrow t_1, t_2, \ldots t_n$:

$$\begin{array}{r} bad_map(task(x), task(y)) \leftarrow taskmap(x, z), \\ taskmap(y, z), constraint(x, y) \end{array} \tag{10}$$

Clause 10 searches for tasks $x$ and $y$ that are mapped onto the same processor $z$ and have a resource constraint. The variables $x, y, z$ are replaced with constants whenever a match is found. Each time a match is found a new term $bad_map(task(x), task(y))$ is derived. These $bad_map$ terms "mark" which tasks have been incorrectly assigned to processors. Consider the instance in Figure 7. The above clause will find two matches: $x = 1, y = 2, z = 3$ and $x = 4, y = 5, z = 6$. The terms $bad_map(task(1), task(2))$ and $bad_map(task(4), task(5))$ will be derived.

In our framework syntactic rules are expressed through Horn clauses, which are used to derive properties of syntax. This process is aided by additional function symbols called *property symbols*, such as the $bad_map$ symbol. Property symbols do not encode syntax, but are used to remember properties of syntax. Figure 8 shows how these components are specified with FORMULA. Line 7 introduces the property symbols and types. (In this case, the property symbols take arguments of any type, i.e. $\top$.) Lines 9 - 12 assert that the endpoints of $taskmap$ and $constraint$ terms exist, even

if they are not explicitly defined. Lines 15 - 16 describe the $bad_map$ axiom. Line 14 deduces a $no_map$ term whenever there is a task $x$ that is *not* mapped to a processor $y$. These clauses are placed inside of a *clauses* block (Line 8).

The negation used in Line 14 is not Boolean negation, but a non-classical form of negation called *negation-as-failure* (NAF) [53]. The expression $\neg t$ is true if it is impossible to derive the term $t$ for certain bindings of variables. A *positive variable* is a variable that occurs in a non-negated term. In line 14 the variable $x$ is positive and the variable $y$ is negative. The rule for evaluation is: (1) Consider each binding of positive variables that makes all non-negated terms true. (2) For each positive binding, check if all the bindings of negative variables fail to derive the negated term. Consider the evaluation of $no_map$ axiom on Figure 7. If $x = 1$, then the assignment of the negative variable $y = 3$ matches the term $taskmap(1, 3)$ and the tail is not satisfied. However, if $x = 7$, then there is no assignment of $y$ so that $taskmap(7, y)$ matches. In this case a term $no_map(task(7))$ is derived indicating that task 7 has not been mapped to a processor. NAF is a powerful extension to Horn logic that allows patterns to be "negated" in an intuitive way. The particular style of Horn logic supported by our tools is *non-recursive Horn logic with stratified negation*, which always terminates.

### 3.2 Domains

Bringing these concepts together, a *domain* $D$ is a structure consisting of: (1) a set of primitive types $P$ over a signature $\Upsilon_P$ encoding the possible syntactic instances, (2) a set of property types $R$ over a signature $\Upsilon_R$ for remembering properties of syntax, (3) and a set of extended Horn clauses $\Theta$ for deriving syntactic properties.

$$D = \langle \Upsilon_P, P,\ \Upsilon_R, R, \Theta \rangle \quad (11)$$

We assume that a fixed order-sorted alphabet $\Sigma_\subseteq$ is common to all domains.

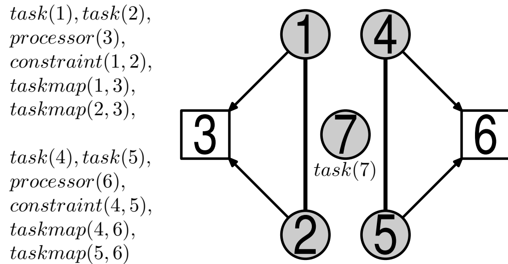



**Figure 7. Syntactic instance with bad task assignments and an unmapped task.**

```
TaskMap Specification - Part 2

 7: properties { no_map(⊤), bad_map(⊤,⊤) }
 8: clauses {
 9:   task(x) ← taskmap(x,y);
10:   processor(y) ← taskmap(x,y);
11:   task(x) ← constraint(x,y);
12:   task(y) ← constraint(x,y);
13:
14:   no_map(task(x)) ← task(x), ¬taskmap(x,y);
15:   bad_map(task(x),task(y)) ← taskmap(x,z),
16:             taskmap(y,z), constraint(x,y);
17: } ...
```

**Figure 8. Axioms of the TaskMap language.**

Previously we showed that domains can be automatically extracted from metamodels and other modeling artifacts [27]. In this paper we do not focus on converting metamodels, feature diagrams, and platforms to domains, but study how domains can be used once this translation is accomplished.

In order to analyze domains there must exist a standard way to determine the legal syntactics instances. We provide two very simple mechanisms for this: Introduce a property symbol $wellform(\cdot)$ so that an instance satisfies invariants if any $wellform$ term can be derived. Otherwise, introduce a $malform(\cdot)$ symbol so that an instance satisfies invariants if it is impossible to derive any $malform$ terms. Domains defined with $wellform$ ($malform$) are called *positive* (*negative*) domains. We use the notation $models(D)$ to indicate the set of syntactic instances satisfying these general rules. If $D$ is a positive domain, then $models(D)$ is the set of syntactic instances that derive $wellform$. If $D$ is a negative domain, then $models(D)$ is the set of instances that do not derive any $malform$ terms.

Line 18 of the TaskMap specification uses the keyword malformed to declare the domain as negative (Figure 9) . This causes the $malform$ symbol to be automatically defined. All that remains is to link the existing invariants with the $malform$ symbol. Two clauses are introduced within the malformed block that derive $malform$ terms whenever there is a bad task assignment or when a task has not been mapped to a processor. This concludes the specification of the TaskMap language.

### 3.3 Model Generation

In the remainder of this paper we show how rich syntax, in the form of domains, can be used across modeling styles for many purposes. Our methods rely on a theorem proving technique called *model generation* [26] (or *abduction* [18]) for non-recursive Horn logic with stratified negation. The

```
TaskMap Specification - Part 3

18: malformed {
19:   malform(no_map(x)) ← no_map(x);
20:   malform(bad_map(x,y)) ← bad_map(x,y);
21: }}
```

**Figure 9. Linking invariants to the $malform$ symbol.**

```
Scheduling a Three-Path

1: proof 3Path of domain TaskMap {
2:   goal {
3:     task(x), task(y), task(z),
4:     constraint(x,y), constraint(y,z),
5:     ¬malform(m)
6:   }}
```

**Figure 10. Formulating a scheduling problem as rich syntax**

technique works as follows: Fix a domain $D$, then a *goal* $G$ is a set of terms with variables. The model generation problem is to construct a set of primitive terms $X$ so that the goal $G$ can be derived from $X$ using the Horn clauses of the domain $D$. In other words, construct a concrete syntactic instance $X$ with the properties $G$, or report that no such $X$ exists.

Model generation is a powerful procedure, as illustrated by the TaskMap language. Let the goal $G$ be a conflict graph with $n$ tasks, and then find a syntactic instance $X$ that satisfies the goal and is also well-formed. This problem is at least as hard as the $n$-coloring problem, because the procedure must determine the number of colors $k$ (i.e. processors) needed to color a conflict graph of $n$ tasks and produce a specific coloring. We have implemented a unique model generation procedure in FORMULA that solves problems expressed over rich syntax. Figure 10 shows an example. Line 1 asks the prover to find a solution for the TaskMap domain. The goal is a conflict graph with three tasks in a path, i.e. three-path. Line 3 requires three tasks $x, y, z$ and Line 4 introduces the conflict edges between $x, y$ and $y, z$. Finally, Line 5 requires that $malform$ cannot be derivable from the solution.

Figure 11 shows one of the solutions generated by FORMULA. Notice that the variables $x, y, z$ have been assigned specific identifiers. Additionally, the new identifiers $4, 5$ have been introduced to represent the processors. In this case, FORMULA produces an optimal solution by coloring the three tasks with two processors. FORMULA creates

$task(1), task(2), task(3)$
$constraint(1, 2),$
$constraint(2, 3),$

$processor(4), processor(5),$
$taskmap(1, 4),$
$taskmap(2, 5),$
$taskmap(3, 4),$

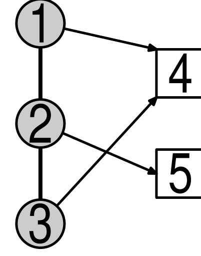



**Figure 11. Syntactic instance generated by** FORMULA

these solutions by reducing a goal/domain pair $(G, D)$ to a Boolean formula $\phi_G(\overline{x})$, where $\overline{x}$ is a set of Boolean variables. A satisfying assignment $s : \overline{x} \rightarrow \mathbb{B}$ is an assignment of Boolean variables to truth values that makes $\phi_G$ true. Each satisfying assignment $s$ corresponds to a set of syntactic instances $\{X_1, X_2, \ldots\}$ that all satisfy the original goal $G$. We denote this set as $[\![s]\!] \subseteq S(P)$. By reducing the problem to a Boolean formula, the many possible solutions are compactly represented and minimal solutions can be systematically discovered.

## 4 Composition of Syntaxes

Composing syntaxes is the process of building larger syntaxes from smaller ones. This process is important in model-based development, because rich syntaxes correspond to subproblems of the overall design problem. For example, the TaskMap language represents the problem of scheduling tasks onto processors. Similarly, the ECU/Bus language (Figure 2) represents the problem of architecting a hardware substrate. Both of these problems must be solved to construct a complete system, and this requires a joining together of the rich syntaxes.

There already exists mechanisms for composing syntaxes, which are well-known to the model-based community. Many composition mechanisms take their inspiration from modern programming languages, borrowing concepts like *namespaces* and *interfaces* to tie together pieces of syntax. The *package merge* mechanism of UML 2.0 supports many styles of composition, including these. Our goal is neither to define the best mechanism for syntax composition, nor to provide a laundry-list formalizing all existing mechanisms. Instead, we provide reusable analysis techniques that determine the properties of composed domains regardless of how one arrives at the composition.

In order to discuss syntax composition we must develop some basic notation for operating on domains. Previously, we assumed that all domains are defined over a fixed order-sorted alphabet $\Sigma_{\subseteq}$. Similarly, we fix an (infinite) universe of function symbols $\mathcal{V}$ over which signatures are defined.

Every domain must choose some finite set of function symbols from $\mathcal{V}$, and we assume that function symbols and constants are disjoint: $\mathcal{V} \cap \Sigma = \emptyset$. A fixed $\mathcal{V}$ and $\Sigma_{\subseteq}$ can be used to construct a well-defined set of domains. Let $\mathbb{D}(\mathcal{V}, \Sigma_{\subseteq})$ be the set of all domains that can be constructed using $\mathcal{V}$ and $\Sigma_{\subseteq}$. Two interesting subsets exist:

$$\mathbb{D}^{+}(\mathcal{V}, \Sigma_{\subseteq}) = \{D \in \mathbb{D}(\mathcal{V}, \Sigma_{\subseteq}) \mid wellform(\cdot) \in \Upsilon_R\}$$
$$\mathbb{D}^{-}(\mathcal{V}, \Sigma_{\subseteq}) = \{D \in \mathbb{D}(\mathcal{V}, \Sigma_{\subseteq}) \mid malform(\cdot) \in \Upsilon_R\}$$

We write $\mathbb{D}, \mathbb{D}^{+}, \mathbb{D}^{-}$ as short-hand for (1) the set of all domains, (2) the set of all positive domains, and (3) the set of all negative domains parametered by $\mathcal{V}$ and $\Sigma_{\subseteq}$. These are also called *small categories,cattheory* in category theory[1]. Each syntactic composition mechanism is represented as a (partial) binary operation *op* over domains $op : \mathbb{D}^2 \rightarrow \mathbb{D}$. We present two useful operators called *structural union* ($\sqcup$) and *product* ($\otimes$).

## 4.1 Structural Union

Structural union combines two domains into one by combining signatures, types, and axioms. In this sense it is similar to the C++ *#include* directive or the Java *import* statement, allowing flexible compositions but without providing any guarantees about the properties of the composition. Structural union is defined as:

$$D \sqcup D' = \left\langle \begin{array}{c} \Upsilon_P \cup \Upsilon'_P, P \cup P', \\ \Upsilon_R \cup \Upsilon'_R, R \cup R', \\ \Theta \cup \Theta' \end{array} \right\rangle \tag{12}$$

Structural union is a partial function over domains, because $D \sqcup D'$ may yield an object that is not a domain. For example, it may yield an object with clauses $\Theta''$ that do not meet the stratification restrictions. Even when the structural union of two domains is defined, it may produce a domain with no syntactic instances, i.e. $models(D'') = \emptyset$. In FORMULA we also write:

**domain *D'* includes *D***

to indicate that domain $D'$ contains the signatures, types, and axioms of $D$ in addition to the contents of $D'$.

## 4.2 Product

A *namespace* is a mechanism for avoiding accidental interactions between classes with the same names. A namespace effectively changes the name of some class Bar to a new fully qualified name Foo.Bar where Foo is the namespace. Another class with the same name Bar, which is not in the namespace Foo, will never be confused with the class Foo.Bar. The intuition behind namespaces can be generalized to compose syntaxes without incurring unexpected interactions. We call this the *product composition*, written $D \otimes D'$, which has the following property:

$$models(D \otimes D') \cong models(D) \times models(D') \tag{13}$$

The product composition produces a rich syntax that is in one-to-one correspondence with the Cartesian product of the two smaller syntaxes. Mathematically, the product composition is a canonical construction for the category of domains; it exists independently of UML or any other standard rules for composition.

We now provide one construction for the product composition, though all constructions satisfying Equation 13 are equivalent up to isomorphism. Given two domains $D, D' \in \mathbb{D}^{-}$ define a pair of renaming functions $(r, r')$ from $\mathcal{V}$ to $\mathcal{V}$ such that the functions are one-to-one, and the symbols from $D$ are renamed to disjoint symbols from $D'$, i.e.

$$r(\Upsilon_P \cup \Upsilon_R) \cap r'(\Upsilon'_P \cup \Upsilon'_R) = \emptyset. \tag{14}$$

Also, the $malform$ symbol is renamed:

$$malform \notin \{r(malform), r'(malform)\}. \tag{15}$$

The renaming maps $r$ and $r'$ act like namespaces by renaming the function symbols so they are disjoint. The product domain is then:

$$D \otimes D' = \langle \Upsilon''_P, P'', \Upsilon''_R, R'', \Theta'' \rangle, \tag{16}$$

$$\Upsilon''_P = r(\Upsilon_P) \cup r'(\Upsilon'_P), \tag{17}$$
$$P'' = r(P) \cup r'(P') \tag{18}$$
$$\Upsilon''_R = r(\Upsilon_R) \cup r'(\Upsilon'_R) \cup \{malform(\cdot)\} \tag{19}$$
$$R'' = r(R) \cup r'(R') \cup \{malform(\top'')\} \tag{20}$$
$$\Theta'' = r(\Theta) \cup r'(\Theta') \cup \left\{ \begin{array}{l} malform(x) \leftarrow r(malform(x)), \\ malform(y) \leftarrow r'(malform(y)) \end{array} \right\} \tag{21}$$

The equations assume the renaming maps are lifted to rename the function symbols appearing in terms and clauses, and each domain has its own top type [2]. Equation 21 ensures that the malformed models of the smaller domains are also the malformed models of the product domain.

## 4.3 Properties of Compositions

Compositions may yield unexpected results due to interactions between invariants. The model generation procedure can be used to check if a composition is sensible. For

[1] The objects are domains, and the morphisms are total functions over well-formed syntactic instances, i.e. $\varphi : models(D_i) \rightarrow models(D_j)$.

[2] Note that each negative domain $D_i$ contains the empty set in $models(D_i)$. This result is needed so that projections exist from the product.

example, discovering a well-formed syntactic instance is a proof that no contradictions were introduced in the composition, i.e. $models(D\ op\ D') \neq \emptyset$. The $IsNotEmpty(D)$ goal proves that an arbitrary (composite) domain is non-empty using model generation. The goal $IsNotEmpty(D)$ for a domain $D$ is given by:

$$IsNotEmpty(D) = \begin{cases} \{wellform(x)\} & \text{if } D \in \mathbb{D}^+ \\ \{\neg malform(x)\} & \text{otherwise.} \end{cases} \tag{22}$$

If $D$ is a positive domain, then construct an instance that derives a $wellform$ term. Otherwise, construct an instance that does not derive any $malform$ terms. If no model can be constructed satisfying $IsNotEmpty(D)$, then the domain is empty and contains a contradiction.

Model generation can also be used to prove relationships between domains. If a domain $D'$ includes domain $D$, then the intent could be to: *Restrict* domain $D$, i.e. $models(D') \subseteq models(D)$, or *extend* domain $D$, i.e. $models(D) \subseteq models(D')$. This can be tested with renaming functions $(r, r')$ that only rename the property symbols $\Upsilon_R$ and $\Upsilon'_R$. Let $D'' = r(D) \sqcup r(D')$, where $r(X)$ is shorthand for applying the renaming function to the signatures, types, and clauses of domain $X$. Assuming $D''$ is defined for structural union, then the goal $IsNotSubset(D, D')$ is given by:

$$IsNotSubset(D, D') = \begin{cases} \left\{ \begin{array}{c} r(wellform(x : \top_D)), \\ \neg r'(wellform(y : \top_{D'})) \end{array} \right\} & \text{if } D, D' \in \mathbb{D}^+ \\ \\ \left\{ \begin{array}{c} r'(malform(x : \top_{D'})), \\ \neg r(malform(y : \top_D)) \end{array} \right\} & \text{if } D, D' \in \mathbb{D}^-. \end{cases} \tag{23}$$

where $x : \top_D$ denotes a variable $x$ whose values range over the terms of domain $D$ (i.e. top type of $D$); similarly $y : \top_{D'}$ ranges over terms of $D'$. (This annotation is not required, but makes it clearer that $r(wellform(x : \top_D))$ renames the $wellform$ function symbol from domain $D$.) The $IsNotSubset(D, D')$ goal attempts to construct a syntactic instance in $D$ that is not in $D'$. If this goal succeeds, then $models(D) \not\subseteq models(D')$, otherwise a subset relationship holds. Using this test, we may also write in FORMULA:

**domain** *D* **restricts** *D'*

to require that $models(D) \subset models(D')$. Alternatively,

**domain** *D* **extends** *D'*

requires $models(D') \subset models(D)$.

## 4.4 Building the CarTask Domain

We put these techniques into practice to support platform-based design in the automotive domain. The first step is to specify the *CarWare* DSL (Figure 2). Figure 12 shows the primitive and property signatures for the CarWare DSL. The Car, ECU, and Msg concepts are encoded as $(n+1)$-ary function symbols, where $n$ is the number of attributes (fields) per class in the metamodel. The first argument of each function symbol is an identifier. The relational concepts CAN and FlexRay are encoded as binary function symbols over identifiers. Bus is an abstract class in the metamodel, so it becomes a property symbol instead of a primitive. Finally, containment is captured by the binary symbol $contains(\cdot, \cdot)$. A term $contains(x, y)$ indicates that $x$ contains $y$. In this example, $x$ is always an identifier, but $y$ may be an identifier or a complex term such as $can(u, v)$.

The CarWare domain is defined by several key clauses, as shown in Figure 13. Lines 11-13 derive the legal forms of containment through the $cancon$ (cancontain) symbol. Lines 14-15 relate the concrete $can$ and $flex$ terms to the abstract $bus$ terms. Figure 14 lists the $malform$ clauses for this domain. Line 17 requires at least one ECU per Car. Line 18 disallows Messages that are neither sent nor received. Lines 19-20 require the endpoints of Buses to exist. Finally, Line 21 checks that all containments are proper.

The *CarTask* domain combines the TaskMap and CarWare domains. Both structural union and product operations are used to carefully build the composition, as shown in Figure 15. In Line 1 the product of TaskMap and CarWare is constructed. By default, a function symbol $f(\ldots)$ from domain $D$ is renamed to $D.f(\ldots)$ in the product. (This preserves the namespace metaphor.) The product composition is an ideal starting point, because no interactions can occur between the domains. Next, the CarTask domain includes the product composition and adds additional clauses. The include mechanism is similar to structural union. These extra clauses bind to the two domains together through the $processor$ and $ecu$ function symbols. Lines 3-4 require a $processor(x, \ldots)$ term for every $ecu(x)$ term; Lines 6-7 require a corresponding $ecu(x)$ term for ev-

CarWare Specification - Part 1

```
1: domain Carware {
2:   primitives {
3:     car(ID, String, String),
4:     ecu(ID, Bool, PosInt),
5:     msg(ID, PosInt, Bool, Bool),
6:     flex(ID, ID), can(ID, ID),
7:     contains(ID, ⊤) }
8:   properties { bus(ID, ID), cancon(ID, ⊤) }...
```

**Figure 12. CarWare primitive and property signatures.**

```
CarWare Specification - Part 2

10: clauses {
11:    cancon(x, y) ← car(x, x1, x2), ecu(y, y1, y2);
12:    cancon(x, bus(y,z)) ← car(x, x1, x2), bus(y, z);
13:    cancon(x, y) ← ecu(x, x1, x2), msg(y, y1, y2, y3);
14:    bus(x, y) ← can(x, y);
15:    bus(x, y) ← flex(x, y); } . . .
```

**Figure 13. Extra clauses for CarWare syntax.**

```
CarWare Specification - Part 3

16: malformed {
17:    malform(x) ← car(x, x1, x2), ¬ecu(y, y1, y2);
18:    malform(x) ← msg(x, x1, false, false);
19:    malform(bus(x,y)) ← bus(x, y), ¬ecu(x, x1, x2);
20:    malform(bus(x,y)) ← bus(x, y), ¬ecu(y, y1, y2);
21:    malform(contains(x,y)) ← contains(x,y),
22:                                   ¬cancon(x,y);
23: }}
```

**Figure 14. Malformed-ness rules for CarWare syntax.**

ery $processor(x, \ldots)$ term. This composition strategy produces an elegant definition for the CarTask domain.

The CarTask domain can be examined using model generation. First, we prove that the composition does not contain contradictions. Evaluating $IsNotEmpty(CarTask)$ yields the following well-formed model in the composition:

$$\left\{ \begin{array}{c} CarWare.car(0, \text{“}a\text{”}, \text{“}b\text{”}), CarWare.ecu(2, \texttt{false}, 3), \\ CarWare.contains(0, 2), TaskMap.processor(2) \end{array} \right\}$$

The CarTask domain should be a restriction of the product. The goal:

$$IsNotSubset(TaskMap \otimes CarWare, CarTask)$$

succeeds by producing a model that is in the product, but not in CarTask:

$$\{\ TaskMap.processor(0)\ \}$$

However, the goal:

$$IsNotSubset(CarTask, TaskMap \otimes CarWare)$$

fails, because every CarTask instance is also an instance of the product. This confirms that indeed:

$$models(CarTask) \subset models(TaskMap \otimes CarWare)$$

In conclusion, disciplined composition operators combined with model generation provides a framework to rigorously reuse rich syntaxes occurring in model-based development.

```
CarTask Specification

1: domain CarTask includes (TaskMap ⊗ CarWare) {
2:  malformed {
3:    malform(CarWare.ecu(x, x1, x2)) ←
4:     CarWare.ecu(x, x1, x2), ¬TaskMap.processor(x);
5:
6:    malform(TaskMap.processor(x)) ←
7:     TaskMap.processor(x), ¬CarWare.ecu(x, x1, x2) ;
8: }}
```

**Figure 15. CarTask syntax is a restriction of the product.**

# 5 Transformations and Validation

Rich syntaxes can also be related through translators; a translator $\tau : models(D) \rightarrow models(D')$ takes a syntactic instance from domain $D$ and translates it into a syntactic instance in domain $D'$. In traditional programming languages a translator may be a compiler that generates machine code from C++. In model-based development translators, also called *model transformations*, are used to: (1) Attach behavioral semantics to rich syntaxes [12] (i.e. semantic anchoring), (2) weave collections of specifications into a single whole [23] (i.e. aspect/model weaving), (3) and migrate models between platforms/abstraction layers. Transformations can be constructed from parts of other transformations, providing a natural mechanism for reuse. For example, the transformation $\tau(X)$:

$$\tau(X) = \tau_3(\tau_2(\tau_1(X)))$$

is $\tau_3$ after $\tau_2$ after $\tau_1$.

## 5.1 Transformations

In our framework transformations are also described using Horn logic. A transformation $\tau$ is:

$$\tau = \langle D, D', \Upsilon_H, H, \Theta_\tau \rangle \tag{24}$$

1. $D$ is the input domain, and $D'$ is the output domain; domains must have disjoint function symbols.
2. $\Upsilon_H$ is a set of *helper* function symbols (disjoint from those in $D, D'$) used by the transformation; $H$ is a set of types over these symbols.
3. $\Theta_\tau$ is a set of Horn clauses that examine input terms from $D$ and deduce output terms in $D'$.

A transformation is executed by providing a set of input terms $X$ from domain $D$. The transformation clauses $\Theta_\tau$ deduce more terms until a fixed-point $X^*$ is reached[3]. Fi-

[3] $X^*$ is the result of forward chaining from $X$ using the clauses $\Theta_\tau$.

```
Simple Transformation

1: transformation Simple from CarWare to TaskMap {
2:    helpers {}
3:    clauses { out.processor(x) ← in.ecu(x,y,z); }
4: }
```

**Figure 16. Transformation with FORMULA**

nally, terms not in the output domain $D'$ are dropped from $X^*$ to yield $X'$:

$$X' = X^* \cap \mathcal{T}_{\Upsilon'_P} \tag{25}$$

For example, let $D = CareWare$, $D' = TaskMap$, $\Upsilon_H = H = \emptyset$, and $\Theta_\tau = \{processor(x) \leftarrow ecu(x, y, z)\}$. This transformation takes a network of ECUs from the CarWare syntax and produces a set of processors in the TaskMap syntax with the same identifiers. Figure 16 shows the simple notation for writing this transformation. Line 1 declares that the transformation Simple has input domain CarWare and output domain TaskMap. The helpers block (Line 2) contains additional function symbols/types needed to support the transformation, i.e. $\Upsilon_H$ and $H$. In this example no helper symbols are needed. Line 3 contains the single clause that produces a $processor$ term for each $ecu$ term. By default, the function symbols of the input and output domains are renamed with in and out to guarantee disjointness.

## 5.2 Structure Preserving Transformations

Transformations, like any other modeling artifact, may contain mistakes. An incorrect transformation may manifest itself by converting some well-formed input (w.r.t. $D$) to a malformed output (w.r.t. $D'$). In this case the transformation does not respect the rules of the rich syntax, which is a serious error. However, this property is not easily established if the input/output domains are rich syntaxes. For example, any transformation into the TaskMap domain must always yield a well-colored set of tasks. In the Simple transformation this is true because every output contains zero tasks, which is trivially well-colored. However, for more complex transformations this fact is not obvious.

Formally, a transformation $\tau$ characterizes a map:

$$[\![\ ]\!]^\tau : \mathcal{P}(\mathcal{T}_{\Upsilon_P}) \to \mathcal{P}(\mathcal{T}_{\Upsilon'_P}) \tag{26}$$

between the powersets of the term algebras of the input and output domains. We say that a transformation $\tau$ is *structure preserving* (SP) if:

$$\forall m \in models(D), \quad [\![m]\!]^\tau \in models(D'). \tag{27}$$

In other words, every well-formed instance in $D$ is mapped to a well-formed instance in $D'$. Checking $\tau$ for the SP property is a form of *validation*; this guarantees that some properties are correct. However, there may be other errors not manifested at the syntactic level, which require additional techniques to discover [59, 30].

A model generation procedure can determine if a transformation is not structure preserving. Transformations that are not SP will have some well-formed input that is malformed under the transformation. It is possible to search for such an input by constructing an *analysis domain* $D(\tau)$:

$$D(\tau) = \langle \emptyset, \emptyset, \Upsilon_H, H, \Theta_\tau \rangle \sqcup (D \otimes prop(D')) \tag{28}$$

$D(\tau)$ is formed by constructing a modified product of the input/output domains, and then combining this with the helper symbols and transformation clauses. The modified product changes the primitive symbols from $D'$ into property symbols:

$$prop(D') = \langle \emptyset, \emptyset, \Upsilon'_P \cup \Upsilon'_R, P' \cup R', \Theta' \rangle. \tag{29}$$

This construction assumes that the renaming maps $r$, $r'$ used in the product have been applied to the transformation clauses $\Theta_\tau$. In the next section, we show how this is represented in FORMULA.

The following goal checks if a transformation $\tau$ is not structure preserving; it is evaluated against $D(\tau)$:

$$IsNotSP(\tau) = \left\{ \begin{array}{ll} \left\{ \begin{array}{c} r(wellform(x : \top_D)), \\ \neg r'(wellform(y : \top_{D'})) \end{array} \right\} & \text{if } D, D' \in \mathbb{D}^+ \\ \left\{ \begin{array}{c} r'(malform(x : \top_{D'})), \\ \neg r(malform(y : \top_D)) \end{array} \right\} & \text{if } D, D' \in \mathbb{D}^-. \end{array} \right. \tag{30}$$

This goal is almost exactly the $IsNotSubset$ goal (Equation 23), except that all the function symbols in the output domain have been converted to property symbols in the analysis domain $D(\tau)$. This forces the model generation procedure to use the transformation rules $\Theta_\tau$ to reason about how malformed models of the output domain can be constructed. If this goal succeeds, then the procedure constructs a well-formed input set $X$ that is malformed after $\tau$ is applied.

## 5.3 Example: Features

Features identify slices of a system that are not isolated from each other. Though feature diagrams capture the legal combinations of features, the engineer must relate each feature to a part of the implementation. In this section we show that features can be defined as transformations from a feature language to an implementation syntax. Transformations that are not structure preserving correspond to badly specified features.

Figures 17, 18 shows several features corresponding to slices of an automotive embedded system. This example

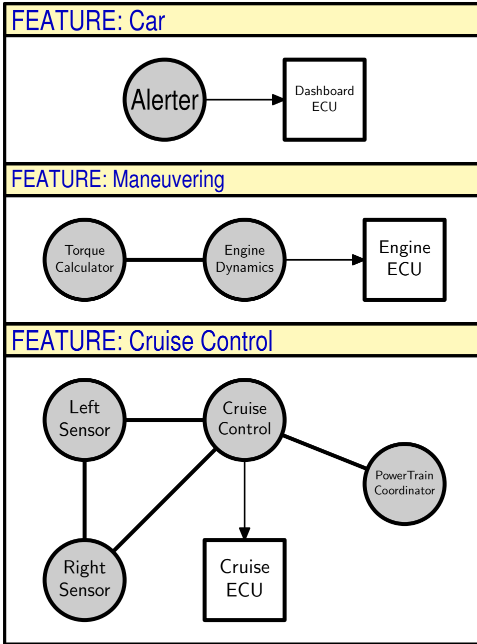



**Figure 17. Relationships between features and implementation.**

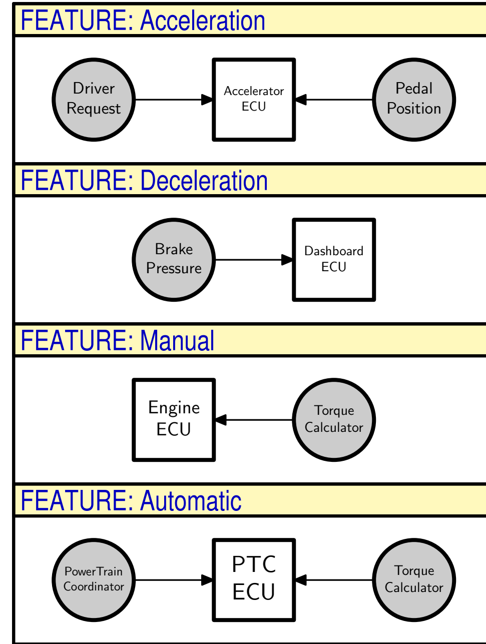



**Figure 18. More features and implementation.**

is adapted from specifications developed by the *AUTOSAR* initiative, which is a consortium whose goal is a standard automotive embedded systems platform [4]. Briefly, The Car feature requires a DashboardECU which hosts the Alerter task. This task controls indicators on the dashboard that alert the driver to problems. The Maneuvering feature requires an EngineECU that hosts the EngineDynamics task. This task monitors the state of the engine, e.g. locations of pistons and temperature. The TorqueCalculator task, which must be located on a different ECU, consumes the engine data to calculate the torque generated by the engine. Finally, the CruiseControl feature needs a CruiseControl task residing on the CruiseECU. Additionally, a LeftSensor and RightSensor must be available to report exact conditions of the left/right wheels. The PowerTrainCoordinator task takes input from the Cruise Control and affects the state of the engine.

In this example features are both partial specifications and span implementation. For example, the Maneuvering feature introduces a TorqueCalculator, but does not specify where it must reside. Notice how the Automatic feature, which is at the bottom of the feature hierarchy, affects the TorqueCalculator task that was introduced at the top of the feature hierarchy. This is typical of the "cross-cutting" nature of features.

#### 5.3.1 Defining the CarFeat Language

We begin by encoding the feature diagram as a domain, called CarFeat. Figure 19 shows the encoding, which reuses the intuition that a feature diagram is a grammar. The primitive symbol $termin(\cdot)$ names the terminal features and the property symbol $nontermin(\cdot)$ names nonterminal features. Finally, the property symbol $badimplies(\cdot,\cdot)$ is derived whenever feature $x$ implies feature $y$, but $y$ is not in the feature set. Line 2 introduces a new finite alphabet called `F` that enumerates the feature names. In the interest of space, the Deceleration feature is treated as terminal.

The clauses of the domain closely resemble a set of grammar productions. The Automatic and Manual features are mutually exclusive; two clauses encode this property (Lines 11-14). Implications between features are easy to capture with Horn logic. The clause in Line 15 deduces $badimplies$ when the Cruise feature is used without the Autom feature. The nonterminal feature Car is produced whenever the Maneuvering feature is produced with no occurrences of $badimplies$. The entire domain is positively defined, i.e. whenever $nontermin(\texttt{Car})$ is produced, the instance is well-formed (Line 18).

#### 5.3.2 The FeatMap Transformation

The transformation from features to implementation is easily expressed via Horn clauses. For instance, the clauses

```
Car Features Specification

1:  domain CarFeat {
2:    enum F { Car, Maneuv, Cruise,
3:             Accel, Decel, Autom, Manual }
4:    primitives { termin(F) }
5:    properties { nontermin(F), badimplies(F,F) }
6:    clauses {
7:      nontermin(Car) ← nontermin(Maneuv),
8:                       ¬badimplies(x,y);
9:      nontermin(Maneuv) ← nontermin(Accel),
10:                         termin(Decel);
11:     nontermin(Accel) ← termin(Autom),
12:                        ¬termin(Manual);
13:     nontermin(Accel) ← termin(Manual),
14:                        ¬termin(Autom);
15:     badimplies(Cruise,Manual) ←
16:             termin(Cruise), termin(Manual);
17:   }
18:   wellformed { wellform(Car) ← nontermin(Car); }
19: }
```

**Figure 19. An automotive feature language.**

below:

$$task(\texttt{Alerter}) \leftarrow nontermin(\texttt{Car})$$
$$processor(\texttt{DashboardECU}) \leftarrow nontermin(\texttt{Car})$$
$$taskmap(\texttt{Alerter}, \texttt{DashboardECU}) \leftarrow nontermin(\texttt{Car})$$

generate the components associated with the Car feature. As a shorthand, FORMULA allows clauses with the same body to be combined into a single clause:

$$task(\texttt{Alerter}),\ processor(\texttt{DashboardECU}),$$
$$taskmap(\texttt{Alerter}, \texttt{DashboardECU}) \leftarrow nontermin(\texttt{Car}).$$

Figure 20 shows a partial specification on the FeatMap transformation. Lines 4-9 define the nonterminal Maneuvering feature and Lines 11-16 define the terminal Automatic feature. The remaining features are omitted in the interest of space.

Though the FeatMap transformation is easily specified, its correctness depends on the input/output domains. For example, the Cruise feature introduces the PowerTrainCoordinator task without assigning it to a processor, which could indicate a mistake in the specification. However, the Automatic feature always accompanies Cruise, and Automatic does assign PowerTrainCoordinator to the PTC_ECU. Thus, correctness cannot be determined on a per feature basis, but must take into account the constraints on the feature language and the implementation syntax. This is precisely the structure preserving property described earlier.

```
FeatureMap Transformation

1:  transformation FeatMap from CarFeat to TaskMap {
2:    helpers {}
3:    clauses {
4:      out.task(TorqueCalc),
5:      out.task(EngineDyn),
6:      out.processor(EngineECU),
7:      out.constraint(TorqueCalc, EngineDyn),
8:      out.taskmap(EngineDyn, EngineECU)
9:                         ← in.nontermin(Maneuv);
10:
11:     out.task(PowerTrainCoor),
12:     out.task(TorqueCalc),
13:     out.taskmap(PowerTrainCoor, PTC_ECU),
14:     out.taskmap(TorqueCalc, PTC_ECU),
15:     out.processor(PTC_ECU)
16:                        ← in.termin(Autom);
17: ...  } }
```

**Figure 20. Specification of features using a transformation**

```
Proving SP of FeatMap

1: proof SP of transformation FeatMap {
2:    goal { in.wellform(x), out.malform(y) } }
```

**Figure 21. Proof constructs well-formed input that violates SP.**

Figure 21 shows the specification of the SP property with FORMULA. Line 1 declares that the proof SP is with respect to the transformation FeatMap. This causes FORMULA to interpret the proof goal over the analysis domain $D(\tau)$. The proof goal asks for a set of terms that are well-formed for the input domain ($in.wellform(x)$) and malformed in the output domain ($out.malform(y)$). Our example intentionally contained several mistakes; FORMULA detects these mistakes producing:

$$X_1 = \{in.termin(\texttt{Manual}), in.termin(\texttt{Decel})\} \tag{31}$$

and

$$X_2 = \left\{ \begin{array}{c} in.termin(\texttt{Autom}), \\ in.termin(\texttt{Cruise}), \\ in.termin(\texttt{Decel}) \end{array} \right\} \tag{32}$$

It also easy to find out why these instances are not structure preserving. Applying the $FeatMap$ transformation to $X_1$ yields the set of terms $X'_1 = [\![X_1]\!]^{FeatMap}$. FORMULA reports the $malform$ terms derivable from $X'_1$ with respect to the output domain TaskMap. This yields the following

set of terms:

$$\left\{ malform\left( bad_map\left( \begin{array}{c} task(\texttt{TorqueCalc}), \\ task(\texttt{EngineDyn}) \end{array} \right)\right)\right\} \tag{33}$$

indicating that the TorqueCalculator and EngineDynamics tasks violated a scheduling constraint. Similarly, $X_2' = [\![X_2]\!]^{FeatMap}$ derives the following $malform$ terms:

$$\left\{ \begin{array}{c} malform(no_map(task(\texttt{LeftSensor}))), \\ malform(no_map(task(\texttt{RightSensor}))) \end{array} \right\} \tag{34}$$

In this case, the LeftSensor and RightSensor tasks were not assigned to processors.

# 6 Design-Space Exploration

The abstractions and analysis techniques of model-based development are useful for automatically investigating many possible system architectures. This process is called *design-space exploration* (DSE). In order to apply DSE the user must characterize: (1) a set of architectures, called the *design space*, and (2) a fitness function that measures the optimality of a point design. DSE prunes the design space, using the fitness function, to present a near-optimal set of designs. The optimization of complex fitness functions is not unique to model-based design and spans many fields including nonlinear control theory [57], game theory [42], and artificial intelligence [61]. However, the success of DSE depends not only on the optimization techniques, but also on the representation of the design space. It is easy for the size of the space to become so large that it cannot be effectively explored. Existing techniques reduce the size of the space by approximating the set of *all interesting designs* with a set of *syntactically related designs*, thereby making exploration feasible [43, 16].

We use the model generation procedure implemented in FORMULA [26] to compactly represent design spaces defined over a rich syntax. This procedure converts a goal/domain pair $(G, D)$ to a Boolean formula $\phi_G(\overline{x})$, where $\overline{x}$ is a set of Boolean variables. Each solution $s$ to $\phi_G$ corresponds to an interesting instance solving the goal; $\phi_G$ may have an exponential number of solutions (in the size of $\overline{x}$). Thus, the Boolean formulas should be viewed as a compact representation of a space of solutions. The $n^{th}$ point design can be lazily constructed by conjuncting the (Boolean) negation of the previous $n-1$ solutions $s_1, \ldots, s_{n-1}$ to $\phi_G$ and solving $\phi_n$:

$$\phi_1 = \phi_G, \quad \phi_{n>1} = \phi_G \wedge \bigwedge_{1 \le i \le n-1} (\neg s_i) \tag{35}$$

Or, the formulas can be converted to *Boolean decision diagrams* (BDDs) permitting easy enumeration of solutions.

```
CarExplore Specification

1:  domain CarExplore restricts CarTask {
2:    properties { hop1(⊤,⊤), hop2(⊤,⊤,⊤),
3:                 hop3(⊤,⊤,⊤,⊤), tooclose(⊤,⊤,⊤),
4:                 toofar(⊤,⊤) }
5:    clauses {
6:      hop1(x,z), hop1(z,x) ← CarWare.bus(x,z), x ≠ z;
7:      hop2(x,y,z) ← hop1(x,y), hop1(y,z), x ≠ z;
8:      hop3(x,y,w,z) ← hop2(x,y,w), hop1(w,z),
9:          z ≠ x, z ≠ y;
10:
11:     toofar(x,z) ←
12:         TaskMap.processor(x), TaskMap.processor(y),
13:         TaskMap.processor(x), TaskMap.processor(y),
14:         ¬hop1(x,z), ¬hop2(x,y,z), ¬hop3(x,y,w,z);
15:     tooclose(x,y,z) ← hop1(x,y), hop1(x,z), hop1(y,z);
16:   }
17:   malformed {
18:     malform(toofar(x,z)) ← toofar(x,z);
19:     malform(tooclose(x,y,z)) ← tooclose(x,y,z);
20: } }
```

**Figure 22. Restricting the** CarTask **domain for DSE.**

Both of these approaches have been used in design space exploration.

## 6.1 A Design Space Representation

In this section we explore possible architectures for a network of ECUs connected by Buses so that the overall system is (1) fault tolerant, (2) minimizes cabling (physical length of wires), and (3) maximizes throughput. We focus on constructing the design space over which exploration takes place. Techniques for formulating and evaluating a reasonable fitness measure can be found in [16].

The first step towards DSE is a suitable rich syntax for expressing relevant architectures. We have already developed such a rich syntax, called the CarTask domain (Figure 15), which represents networks of ECUs combined with tasks. Given a set of tasks $T$ and processors $P$, the design space should be a set of CarTask instances, each of which contains $T$ and $P$ along with some bus architecture. However, the original CarTask domain did not enforce that ECUs should be connected; so additional clauses must be added to the CarTask domain to define a reasonable design space.

Figure 22 shows the additional clauses needed for DSE over bus topologies. The property symbols $hop1$, $hop2$, $hop3$ record the distance between any two processors in the

Cruise Control Design Space

```
1: proof DS of domain CarExplore {
2:   goal { ¬malform(m) }
3:     with FeatMap ({ termin(Cruise),
4:           termin(Autom), termin(Decel) });
5: }
```

**Figure 23. Constructing a design space**

network. Any two processors $x$ and $z$ that are farther than three hops will not derive any of these properties. Lines 6-9 define hops using *disequality* constraints, which require two variables to take distinct values. Topologies that are too weakly connected or too strongly connected (too much cabling) are characterized with the $tooclose$ and $toofar$ properties. Two processors $x$ and $z$ are too far away if they a farther than three hops (Line 11). Three processors $x, y$, and $z$ are too connected if they form a triangle (Line 15). Line 1 declares that the CarExplore domain must be a proper *restriction* of the CarTask domain. The additional malformedness clauses (Lines 18-19) introduce topology restrictions so there is a restriction relationship. Regardless, FORMULA can check that the models of CarExplore are a subset of the models of CarTask.

Constructing the design space is now a matter of writing a proof in the CarExplore domain. Figure 23 shows a design space for a Car containing the Cruise control and Automatic transmission features. The goal of the proof requires architectures that are syntactically correct with topologies that are neither too sparse nor too dense. Lines 3-4 add to the goal all of the tasks and processors associated with the cruise control configuration. These terms are generated by applying the FeatMap transformation to the input $\{termin(\texttt{Autom}), termin(\texttt{Cruise}), termin(\texttt{Decel})\}$. FORMULA infers the renaming of function symbols when possible.

## 6.2 Details of the Design Space

We now summarize the resulting design space constructed by model generation. The procedure attempts to calculate a finite set of candidate terms $S$ with the property that if there exists any solution to the goal, then there must also exist a solution $S' \subseteq S$. Once the set $S$ is calculated, each term $t_i \in S$ is converted to a Boolean variable $b_i$, and a Boolean formula $\phi_G$ is generated over these variables. If $b_i$ is true in satisfying assignment of $\phi_G$, then the term $t_i$ is in the solution; otherwise $t_i$ is not in the solution.

Intuitively, the set of candidate terms is built in several phases. First, non-negated terms without variables, *called ground terms*, appearing in the goal must be part of any solution; these terms are immediately included in $S$. Table 2 shows some of the ground terms introduced by the FeatMap transformation. The *Variable* column lists the Boolean variable associated with each term. Next, the procedure finds ground terms that must be considered because of negations. For example, the goal $\neg toofar(u, w)$ will be considered for $u = \texttt{CruiseECU}$ and $w = \texttt{EngineECU}$. This causes $hop1(u, w)$ to be considered, resulting in the conclusion that $bus(\texttt{CruiseECU}, \texttt{EngineECU})$ may be in some solution. Table 3 shows ground terms introduced by this process. Notice that these terms include all the possible Buses that may appear in the architecture.

| Ground Term | Variable |
|---|---|
| $processor(\texttt{EngineECU})$ | $a_1$ |
| $processor(\texttt{DashboardECU})$ | $a_2$ |
| $processor(\texttt{CruiseECU})$ | $a_3$ |
| $processor(\texttt{PTCECU})$ | $a_4$ |
| $task(\texttt{LeftSensor})$ | $a_5$ |
| $task(\texttt{RightSensor})$ | $a_6$ |
| $constraint(\texttt{LeftSensor}, \texttt{RightSensor})$ | $a_7$ |

**Table 2. Ground terms appearing in goal.**

| Ground Term | Variable |
|---|---|
| $bus(\texttt{EngineECU}, \texttt{DashboardECU})$ | $b_1$ |
| $bus(\texttt{CruiseECU}, \texttt{EngineECU})$ | $b_2$ |
| $bus(\texttt{EngineECU}, \texttt{PTCECU})$ | $b_3$ |
| $bus(\texttt{PTCECU}, \texttt{DashboardECU})$ | $b_4$ |
| $bus(\texttt{PTCECU}, \texttt{CruiseECU})$ | $b_5$ |
| $bus(\texttt{DashboardECU}, \texttt{CruiseECU})$ | $b_6$ |

| Non-ground Term | Variable |
|---|---|
| $ecu(\texttt{EngineECU}, x_1, y_1)$ | $c_1$ |
| $ecu(\texttt{CruiseECU}, x_2, y_2)$ | $c_2$ |
| $processor(p_1)$ | $c_3$ |
| $processor(p_2)$ | $c_4$ |
| $taskmap(\texttt{LeftSensor}, p_1)$ | $c_5$ |
| $taskmap(\texttt{RightSensor}, p_2)$ | $c_6$ |

**Table 3. Terms implied by negations.**

Model generation may also introduce terms that contain variables. Recall that the Cruise control feature contained tasks LeftSensor and RightSensor that were not assigned to any processor. However, these tasks must be assigned to processors, otherwise $malform(\cdot)$ will be derived. In response, FORMULA introduces new variables $p_1$ and $p_2$ that stand for these missing processors. Table 3 also shows some of the non-ground terms.

After the candidate set $S$ has been constructed, a Boolean formula $\phi_G$ is produced that encodes the proof

goal. In the interest of space, we only sketch $\phi_G$. The ground terms in Table 2 appear in the goal and must be in any solution, leading to the simple encoding:

$$\phi_G^I = a_1 \wedge a_2 \wedge \ldots \wedge a_7 \quad (36)$$

The domain constraints require an $ecu$ term for each $processor$ term, and a $taskmap$ term for each $task$. The encoding forces these terms to appear in pairs:

$$\phi_G^{II} = \begin{bmatrix} (a_1 \wedge c_1) \vee (\neg a_1 \wedge \neg c_1) \end{bmatrix} \wedge \\ \begin{bmatrix} (a_2 \wedge c_2) \vee (\neg a_2 \wedge \neg c_2) \end{bmatrix} \wedge \\ \begin{bmatrix} (a_5 \wedge c_5) \vee (\neg a_5 \wedge \neg c_5) \end{bmatrix} \wedge \\ \begin{bmatrix} (a_6 \wedge c_6) \vee (\neg a_6 \wedge \neg c_6) \end{bmatrix} \wedge \ldots \quad (37)$$

Care must be taken when mapping tasks LeftSensor and RightSensor due to scheduling conflicts:

$$\phi_G^{III} = (p_1 \neq p_2) \vee (\neg a_2 \vee \neg c_5 \vee \neg c_6) \quad (38)$$

This is not strictly a Boolean formula, because $p_1 \neq p_2$ is a disequality over non-Boolean variables. These non-Boolean disequalities can also be reduced to Boolean variables [11]; for simplicity we write the formula in this extended form. Next, buses may be too close:

$$\phi_G^{IV} = (\neg b_1 \vee \neg b_2 \vee \neg b_6) \wedge (\neg b_1 \vee \neg b_3 \vee \neg b_4) \wedge \\ (\neg b_2 \vee \neg b_3 \vee \neg b_5) \wedge (\neg b_4 \vee \neg b_5 \vee \neg b_6) \wedge \ldots \quad (39)$$

Finally, processors must be close enough to each other. This rule generates a large subformula; only a small portion is shown:

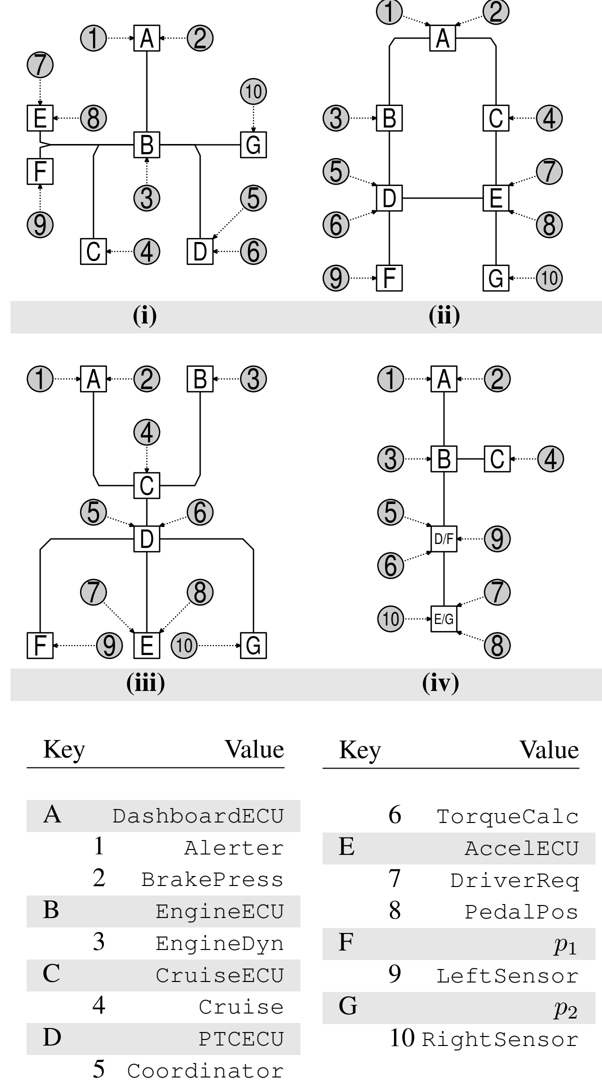
$$\phi_G^V = (a_1 \wedge a_2 \wedge b_1) \vee (a_1 \wedge a_3 \wedge a_2 \wedge b_2 \wedge b_6) \vee \ldots \quad (40)$$

The entire design space is represented by the conjunction of these formulas:

$$\phi_G = \phi_G^I \wedge \phi_G^{II} \wedge \phi_G^{III} \wedge \phi_G^{IV} \wedge \phi_G^V \quad (41)$$

Currently the SMT solver *Z3* [17] is used to evaluate these formulas. Z3 also chooses reasonable values for non-Boolean variables, like processor IDs, when it returns a satisfying assignment.

Figure 24 shows some of the interesting architectures in this design space. The first design is a *star* topology; all messages pass through the EngineECU. The topology of the second design splits the system into two subnetworks with a strategically placed bridge between the PTC and Accelerator ECUs. Without the bridge, some ECUs would be too far from each other. Design three takes a different approach, and splits the network into two *star* topologies with a bridge between the subnetworks. Finally, the fourth design is the most serialized design possible. This architecture is possible because the left and right sensors are scheduled onto existing ECUs, instead of instantiating new ones.




| Key | Value | Key | Value |
|---|---|---|---|
| A | DashboardECU | 6 | TorqueCalc |
| 1 | Alerter | E | AccelECU |
| 2 | BrakePress | 7 | DriverReq |
| B | EngineECU | 8 | PedalPos |
| 3 | EngineDyn | F | $p_1$ |
| C | CruiseECU | 9 | LeftSensor |
| 4 | Cruise | G | $p_2$ |
| D | PTCECU | 10 | RightSensor |
| 5 | Coordinator | | |

**Figure 24. Members of the design space.**

## 7 Related Work

Formalizations and applications of rich syntax have appeared in many different forms. Within the domain-specific language community, graph-theoretic formalisms [19, 8, 48] have received the most research attention. However, the majority of work focuses on graph rewriting systems as a foundation for model transformations. See [40, 33] for a taxonomy of existing graph-theoretic model transformation approaches. The problems of calculating properties of rich syntax, composing syntax with known properties, and constructing design space representations have not received the same attention from graph-theoretic methods. For example, the model transformation tool VIATRA [13] supports executable Horn logic (i.e. Prolog) to specify transformations, but does not focus on restricting expressiveness for the pur-

pose of analysis.

The visibility of UML has driven researchers to formalize it semantics. This is a non-trivial task because UML includes many capabilities (diagrams) including metamodeling, state machines, activities, sequence charts (interactions), and use-case diagrams [47]. Approaches for formalizing UML must tackle the temporal nature of its various behavioral semantics, necessitating more expressive formal methods. Well-known tools/methods such as Alloy [24], B [37], and Z [20] have been used to varying degrees of success. These approaches make trade-offs between expressiveness and the degree of automated analysis. For example, Z and B proofs typically require interactive theorem provers [10, 5] and model generation may not be supported. Z or B formalizations of UML could be a vehicle for studying rich syntax, but automated analysis is less likely to be found.

Alloy, like FORMULA, is less expressive than other methods, thereby supporting automated analysis [25]; it also has a recently improved model generation (model finding) procedure [58]. However, the mathematical underpinnings of Alloy are quite different from FORMULA: Alloy supports first-order logic with relations over atoms plus transitive closure. Contrarily, our framework is based on a non-monotonic extension of Horn logic [26]. One key difference is that FORMULA specifications can be executed like standard *logic programs* [53]. Complexity-theory also offers a coarse-grained way of comparing logic programs with other methods [15].

The BNF grammars of traditional programming languages can be extended to capture richer syntaxes. *Attribute grammars* (AGs) [49] , proposed by Knuth [32], could be the earliest example of such a mechanism. AGs allow the productions of a BNF grammar to trigger actions capable of examining tokens and attaching new data to tokens. These actions can be specified programmatically, thereby significantly increasing the power of the grammar. However, calculating properties of languages specified through AGs depends on the expressiveness of the actions. Additionally, composing AGs has proved to be a difficult task [21]. More recently, *pluggable type systems* have been studied as a mechanism to compose the type systems of traditional programming languages [3].

Tools for creating and editing models (*CRUD* tools) [2, 56] intersect with databases, because they require a persistence layer for storing many different models across many domains. Various extensions of Horn logic have been utilized by the *declarative database* community [41] as powerful query languages; this work fits naturally with our notion of a *domain*. Functional programming has also been used to declaratively operate on databases [35]. The *Language Integrated Query* (LINQ) project extends this work, allowing in-memory data structures to be queried just like databases [39].

Numerous examples of structural representations for design spaces can be found in the literature. For example, [29] performs DSE for arbitrary algorithms by extracting a data dependency graph from the steps of a given algorithm $A$. The design space is the set all similar dependency graphs not contradicting the dependencies of the original algorithm. Dependency graphs form a syntactic construct approximating an ideal design space (consisting of all algorithms $A'$ that compute the same function as $A$). The model-based tool DESERT [43] compactly represents automotive design spaces using AND-OR trees to encode architectural choices. These trees are converted to BDDs allowing the design space to be pruned without explicit enumeration of its elements. Recent work on fitness functions for evaluating automotive design spaces can be found in [16].

# 8 Conclusion

In conclusion, we have shown that Horn logic extended with negation provides a powerful framework for integrating the rich syntaxes and transformations occurring in model-based development. By restricting our focus to a well-understood logic-based kernel, we obtain many important and efficient analysis techniques. Specifically, we provide sound mechanisms for composing syntaxes, determining properties of compositions, detecting mistakes in model transformations, and constructing design spaces over rich syntax. These techniques have been implemented in FORMULA.

# References


[1] F. D. B. A. L. Sangiovanni-Vincentelli, L. Carloni and M. Sgroi. Benefits and challenges for platform-based design. In *Proceedings of the Design Automation Conference (DAC'04)*, June 2004.

[2] A. B. G. K. J. G. C. T. I. G. N. J. S. P. V. A. Ledeczi, M. Maroti. The generic modeling environment. *Workshop on Intelligent Signal Processing*, May 2001.

[3] C. Andreae, J. Noble, S. Markstrum, and T. Millstein. A framework for implementing pluggable type systems. *SIGPLAN Not.*, 41(10):57–74, 2006.

[4] AUTOSAR. Achievements and exploitation of the autosar development partnership. Technical report, 2006.

[5] R. Banach and S. Fraser. Retrenchment and the b-toolkit. In *ZB 2005: Formal Specification and Development in Z and B, 4th International Conference of B and Z Users*, pages 203–221, april 2005.

[6] D. Batory, J. N. Sarvela, and A. Rauschmayer. Scaling stepwise refinement. In *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*, pages 187–197, Washington, DC, USA, 2003. IEEE Computer Society.

[7] D. S. Batory. Feature models, grammars, and propositional formulas. In *SPLC*, pages 7–20, 2005.

[8] J. Bezivin and O. Gerbé. Towards a precise definition of the omg/mda framework. In *In Proceedings of the 16th Conference on Automated Software Engineering*, pages 273–280, 2001.

[9] T. Blickle, J. Teich, and L. Thiele. System-level synthesis using evolutionary algorithms. Technical Report TIK Report-Nr. 16, Gloriastrasse 35, 8092 Zurich, 1996.

[10] A. D. Brucker, F. Rittinger, and B. Wolff. HOL-Z 2.0: A proof environment for Z-specifications. *Journal of Universal Computer Science*, 9(2):152–172, Feb. 2003.

[11] R. E. Bryant and M. N. Velev. Boolean satisfiability with transitivity constraints. *ACM Trans. Comput. Log.*, 3(4):604–627, 2002.

[12] K. Chen, J. Sztipanovits, and S. Neema. Compositional specification of behavioral semantics. In *DATE*, pages 906–911, 2007.

[13] G. Csertán, G. Huszerl, I. Majzik, Z. Pap, A. Pataricza, and D. Varró. Viatra - visual automated transformations for formal verification and validation of uml models. In *ASE*, pages 267–270, 2002.

[14] K. Czarnecki and K. Pietroszek. Verifying feature-based model templates against well-formedness ocl constraints. In *GPCE*, pages 211–220, 2006.

[15] E. Dantsin, T. Eiter, G. Gottlob, and A. Voronkov. Complexity and expressive power of logic programming. *ACM Comput. Surv.*, 33(3):374–425, 2001.

[16] A. Davare, Q. Zhu, M. D. Natale, C. Pinello, S. Kanajan, and A. L. Sangiovanni-Vincentelli. Period optimization for hard real-time distributed automotive systems. In *DAC*, pages 278–283, 2007.

[17] L. de Moura and N. Bjørner. Z3: An Efficient SMT Solver. *In Proceedings of Fourteenth International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2008)*, March 2008.

[18] M. Denecker and D. D. Schreye. Sldnfa: An abductive procedure for abductive logic programs. *J. Log. Program.*, 34(2):111–167, 1998.

[19] H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer. Fundamental theory for typed attributed graphs and graph transformation based on adhesive hlr categories. *Fundam. Inform.*, 74(1):31–61, 2006.

[20] A. Evans, R. B. France, and E. S. Grant. Towards formal reasoning with uml models. In *In Proceedings of the Eighth OOPSLA Workshop on Behavioral Semantics*.

[21] R. Farrow, T. J. Marlowe, and D. M. Yellin. Composable attribute grammars: support for modularity in translator design and implementation. In *POPL '92: Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 223–234, 1992.

[22] A. L. T. B. G. Karsai, J. Sztipanovits. Model-integrated development of embedded software. *Proceedings of the IEEE*, 91(1):145–164, January 2003.

[23] J. G. Gray and S. Roychoudhury. A technique for constructing aspect weavers using a program transformation engine. In *AOSD*, pages 36–45, 2004.

[24] D. Jackson. A comparison of object modelling notations: Alloy, uml and z. Technical report, August 1999.

[25] D. Jackson. Automating first-order relational logic. In *SIGSOFT FSE*, pages 130–139, 2000.

[26] E. K. Jackson and W. Schulte. Model generation for horn logic with stratified negation. In *Proceedings of the 28th International Conference on Formal Techniques for Networked and Distributed Systems*, 2008.

[27] E. K. Jackson and J. Sztipanovits. Towards a formal foundation for domain specific modeling languages. *Proceedings of the Sixth ACM International Conference on Embedded Software (EMSOFT'06)*, pages 53–62, October 2006.

[28] M. Jackson and P. Zave. Domain descriptions. In *Proceedings of the IEEE Conference on Requirements Engineering*, pages 56–64, 1993.

[29] S. Kakita, Y. Watanabe, D. Densmore, A. Davare, and A. L. Sangiovanni-Vincentelli. Functional model exploration for multimedia applications via algebraic operators. In *ACSD*, pages 229–238, 2006.

[30] G. Karsai and A. Narayanan. On the correctness of model transformations in the development of embedded systems. In *Monterey Workshop*, pages 1–18, 2006.

[31] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In M. Akşit and S. Matsuoka, editors, *Proceedings European Conference on Object-Oriented Programming*, volume 1241, pages 220–242. Springer-Verlag, Berlin, Heidelberg, and New York, 1997.

[32] D. E. Knuth. The genesis of attribute grammars. In *In Proceedings of Attribute Grammars and their Applications*, pages 1–12, 1990.

[33] A. Königs and A. Schürr. Multi-domain integration with mof and extended triple graph grammars. In *Language Engineering for Model-Driven Software Development*, 2004.

[34] E. A. Lee and S. Neuendorffer. Actor-oriented models for codesign: Balancing re-use and performance. *Formal Methods and Models for Systems, Kluwer*, 2004.

[35] D. Leijen and E. Meijer. Domain specific embedded compilers. In *DSL*, pages 109–122, 1999.

[36] M. Mannion. Using first-order logic for product line model validation. In *SPLC*, pages 176–187, 2002.

[37] R. Marcano and N. Levy. Using b formal specifications for analysis and verification of uml/ocl models. In *In Workshop on consistency problems in UML-based software development. 5th International Conference on the Unified Modeling Language*, pages 91–105, 2002.

[38] D. Marx. Graph coloring problems and their applications in scheduling. *Periodica Polytechnica Ser. El. Eng.*, 48(1-2):5–10, 2004.

[39] E. Meijer, B. Beckman, and G. M. Bierman. Linq: reconciling object, relations and xml in the .net framework. In *In Proceedings of the ACM SIGMOD International Conference on Management of Data*, page 706, 2006.

[40] T. Mens, P. V. Gorp, D. Varró, and G. Karsai. Applying a model transformation taxonomy to graph transformation technology. *Electr. Notes Theor. Comput. Sci.*, 152:143–159, 2006.

[41] J. Minker. Logic and databases: A 20 year retrospective. In *Logic in Databases*, pages 3–57, 1996.

[42] R. B. Myerson. *Game Theory: Analysis of Conflict*. Harvard University Press, September 1997.

[43] S. Neema, J. Sztipanovits, G. Karsai, and K. Butts. Constraint-based design-space exploration and model synthesis. In *EMSOFT*, pages 290–305, 2003.

[44] Object Management Group. Meta object facility specification v1.4. Technical report, 2002.
[45] Object Management Group. Mda guide version 1.0.1. Technical report, 2003.
[46] Object Management Group. Ocl specification, v2.0. Technical report, 2006.
[47] Object Management Group. Omg unified modeling language (omg uml), superstructure, v2.1.2. Technical report, 2007.
[48] F. Orejas, H. Ehrig, and U. Prange. A logic of graph constraints. In *FASE*, pages 179–198, 2008.
[49] J. Paakki. Attribute grammar paradigms—a high-level methodology in language implementation. *ACM Comput. Surv.*, 27(2):196–255, 1995.
[50] A. Pinto, A. Bonivento, A. L. Sangiovanni-Vincentelli, R. Passerone, and M. Sgroi. System level design paradigms: Platform-based design and communication synthesis. *ACM Trans. Design Autom. Electr. Syst.*, 11(3):537–563, 2006.
[51] C. Prehofer. Feature-oriented programming: A new way of object composition. *Concurrency and Computation: Practice and Experience*, 13(6):465–501, 2001.
[52] A. Pretschner, M. Broy, I. H. Kruger, and T. Stauner. Software engineering for automotive systems: A roadmap. *Future of Software Engineering, 2007. FOSE '07*, pages 55–71, May 2007.
[53] T. C. Przymusinski. Every logic program has a natural stratification and an iterated least fixed point model. In *PODS '89: Proceedings of the eighth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 11–21, 1989.
[54] S. Sastry, J. Sztipanovits, R. Bajcsy, and H. Gill. Scanning the issue - special issue on modeling and design of embedded software. *Proceedings of the IEEE*, 91(1):3–10, 2003.
[55] F. Sethna, E. Stipidis, and F. Ali. What lessons can controller area networks learn from flexray. *Vehicle Power and Propulsion Conference, 2006. VPPC '06. IEEE*, pages 1–4, 6-8 Sept. 2006.
[56] The Eclipse Modeling Framework. www.eclipse.org/emf/. Technical report.
[57] C. Tomlin, J. Lygeros, and S. Sastry. Computing controllers for nonlinear hybrid systems. In *HSCC*, pages 238–255, 1999.
[58] E. Torlak and D. Jackson. Kodkod: A relational model finder. In *In Proceedings of the 13th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2007)*, pages 632–647, 2007.
[59] D. Varró. Automated formal verification of visual modeling languages by model checking. *Software and System Modeling*, 3(2):85–113, 2004.
[60] S. Wang, S. K. Birla, and S. Neema. A modeling language for vehicle motion control behavioral specification. In *SEAS '06: Proceedings of the 2006 international workshop on Software engineering for automotive systems*, pages 53–60, 2006.
[61] L. D. Whitley. An overview of evolutionary algorithms: practical issues and common pitfalls. *Information & Software Technology*, 43(14):817–831, 2001.